Applying XML technologies in Requirements Verification*

Amador Durán Antonio Ruiz Rafael Corchuelo Miguel Toro

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Sevilla, Avda. Reina Mercedes s/n
41012 Sevilla, España
e-mail: {amador,aruiz,corchu,mtoro}@lsi.us.es

Abstract

In this paper, we present an approach for the automatic verification of software requirements specifications. This approach is based on the representation of software requirements in XML and the usage of the XSLT language not only to automatically generate requirements documents, but also to verify some desired quality properties and to automatically compute some defect–predictive metrics. These ideas have been implemented in REM, an experimental requirements management tool that is also described in this paper.

Keywords: Requirements Engineering, Requirements Verification, XML, XSLT

1 Introduction

Quality related activities in the requirements engineering process are basically requirements analysis, requirements verification and requirements validation (see the UML activity diagram of figure 1). In these three activities, requirements evolve from draft to managed, i.e. baselined, eliminating conflicts and defects during their evolution.

During requirements analysis, the main goal is detecting conflicts, which will be handled by the requirements negotiation process, and gain a deeper understanding of the requirements. The usual technique for achieving this goal is conceptual modeling [15].

During requirements verification, and paraphrasing Boehm [6], the goal is answering the question "*Am I building the requirements right?*". More formally, Pohl [19] defines the goal of requirements verification as checking the requirements according to desired quality properties. Some of these requirements quality properties have to do with requirements semantics but others have to do with syntactic, structural or pragmatic aspects of requirements (see [13] for a complete classification of quality properties of requirements). The lack of the semantic properties causes *knowledge* errors whereas the lack of non–semantic properties causes *specification* errors, as described in [12].

On the other hand, the goal of requirements validation is also defined by Boehm as answering the question "*Am I building the right requirements?*". Pohl defines it as certifying that the requirements are consistent with the intentions of customers and users.

Verification of semantic properties of requirements is closely related to requirements validation and requires human participation, whereas verification of non–semantic properties should be as automated as possible. As a matter of fact, distinction between requirements verification of semantic properties and requirements validation is sometimes subtle and many authors use both terms interchangeably.

In this paper, we present an automated approach for the verification of some quality properties of requirements and for the automatic computing of some defect–predictive requirements metrics. Most of these properties can be classified as non–semantic, but we have also developed some heuristics to check potential problems

*This work is partially funded by the CICYT project TIC 2000–1106–C02–01 (GEOZOCO) and by the CYTED project VII.18 (WEST).

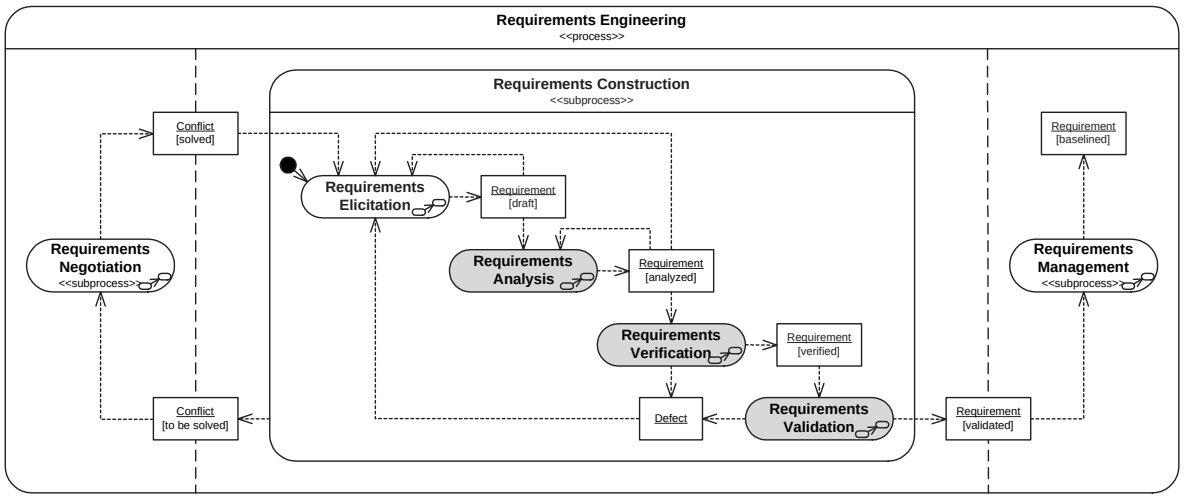

Figure 1: Requirements Engineering Process Model

with some semantic properties. Our approach is based on the emergent technology built around XML [4] and its companion language XSLT [3].

The rest of the article is organized as follows. First, we briefly describe the basics of XML and XSLT needed to understand the following sections. Then, we describe REM, an experimental requirements management tool [10, 11], the XML model of requirements used by REM and how XSLT can be used to verify some quality properties of requirements expressed in XML. Finally, we discuss some related work, present some results and point out future work.

2 XML and XSLT

2.1 XML Basics

There are millions of web pages written in HTML available in Internet. In these web pages, information is mixed with formatting elements, making the automatic processing of information very difficult. XML [4] is a language designed for representing pure information in Internet. Information in XML is represented by *elements*. An XML element is made up of a start tag, an end tag, and other tags or data in between. For example, for representing the information about a book, we might have the following XML element named book:

```
<book isbn="X-XXX-XXXX-X">
  <author>Miguel de Cervantes</author>
  <title>El Quijote</title>
</book>
```

As can be seen, the information about a book is between the <book> and </book> tags and it is easy to parse by a computer program. The author and title elements are considered as children of the book element, thus forming a hierarchy. An XML document must always have one and only one *root element* at the top of its hierarchy.[1]

In order to allow information interchange between two or more parties using XML, they must agree about element grammar and semantics. Element grammar is specified as regular expressions in DTDs (Document Type Definitions) [4]. For example, the DTD fragment for the previous XML data might be the following:

[1] XML organizes information hierarchically, but other data structures like graphs can also be represented by means of ID and IDREF attributes. See [4] for details.

```
<!ELEMENT book (author+,title)>
  <!ATTLIST book isbn ID #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title  (#PCDATA)>
```

where it is stated that a book element can contain one o more author elements and only one title element. An XML element can also have attributes. For example, isbn is defined as a required identification attribute of book, *i.e.* there cannot exist two books with the same value for the isbn attribute in the same XML document. Those elements that contain only text are said to contain #PCDATA, that stands for *parsed character data*.

2.2 Transforming XML

There are many situations in which XML data need to be transformed. For example, for representing the same information according to another set of tags or for presenting XML data in an HTML page. XSLT [3] is a language based on transformation patterns. An XSLT stylesheet, which is also a an XML document, searches for patterns in the XML data and applies programmed transformations, thus generating some output results. For example, if we wanted to show information about books in a web browser, we could apply the following XSLT transformation rule:

```
<xsl:template match="book">
   <b><xsl:value-of select="title"/></b> (ISBN <xsl:value-of select="@isbn"/>)
   was written by <em><xsl:value-of select="author[1]"/></em>
</xsl:template>
```

The informal semantics of this XSLT rule are "when you find a book element, generate its title in boldface, then its ISBN attribute (notice the @ prefix for attributes), and then its first author in emphasized mode". In the XSLT code, text literals like HTML tags can be mixed with element values, which are obtained by means of the xsl:value-of statement. If we applied this XSLT rule to the previous XML data, the result of the transformation would be something like this when rendered in a web browser:

> **El Quijote** (ISBN X-XXX-XXXX-X) was written by *Miguel de Cervantes*

Although there are many more details about XML and XSLT, we think that this brief introduction should be enough for those readers not familiar with XML technologies in order to understand the rest of this paper.

3 REM: An XML–based Requirements Management Tool

REM (*REquirements Manager*) is an experimental requirements management tool developed by one of the authors as part of his PhD. Thesis [10, 11]. In REM, a requirements engineering (RE) project is considered to be composed by four documents:

1. a customer–oriented requirements document, usually containing requirements in natural language expressed in terms of customer's vocabulary, also known as *C–requirements* [8].
2. a developer–oriented requirements document, usually containing requirements models and more technical information, also called *D–requirements* [8].
3. a registry for detected conflicts and negotiation support.
4. a registry for change requests.

In REM, C–requirements and conflicts are expressed in natural language using predefined requirements templates and some linguistic patterns (see [11] for details). For expressing D–requirements, we have chosen a subset of the UML [7] with strong influences from [9].

3.1 REM Architecture

REM documents, *i.e.* RE projects composed by the four documents previously described, are stored in relational light–weight databases. When the user creates a new REM document, the basic structure is taken from a *REM base document* (see figure 2), that can be empty or can contain the mandatory sections of software requirements standards like [1]. Any ordinary REM document can be selected as a base document, so users can create their own base documents or reuse other REM documents.

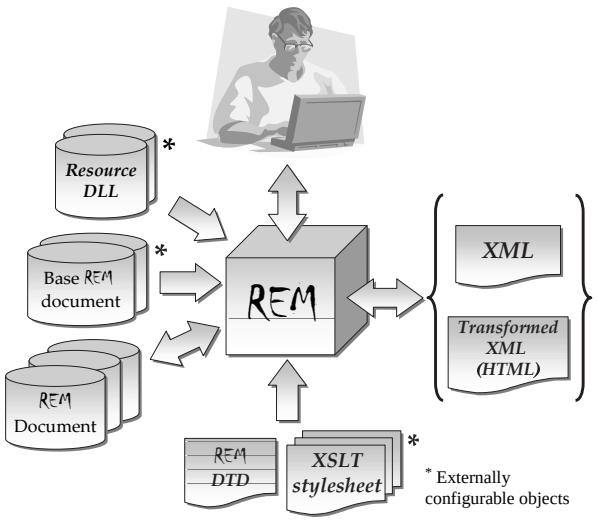

Figure 2: REM Architecture

In order to provide immediate feedback on user actions, REM generates XML data corresponding to the document being edited, applies an external XSLT stylesheet that transforms XML data into HTML and shows the resulting HTML to the user. In this way, whenever the user changes a requirements document, he or she can see the effects immediately.

In a similar way the REM base document can be tailored, the user can also change document appearance by selecting or creating different external XSTL stylesheets. The default XSLT stylesheet generates a highly hyperlinked document, easing navigation of requirements documents (see right side of figure 3).

Other configurable aspect of REM is the language of the user interface. The user can choose it by selecting an external resource dynamic link library (DLL). At the moment of writing, we have developed two external resource DLLs for REM, one in Spanish and other in English. Another one in Portuguese is under development.

3.2 REM User Interface

The user interface of REM presents two different views to the user (see figure 3). On the left, the user can see a tabbed view with four tree views, one for each requirements document in the RE project. On the right hand, the result of the XSLT transformation of the XML data is presented to the user in a embedded web browser.

In any of the four tree views, the user can directly manipulate objects by drag and drop or by context menus. Only actions that have sense can be performed, following a *correct–by–construction* approach, thus increasing quality and saving verification effort. For example, actions of use case steps can be of three different classes (see figure 4): *actor action*, if the action is performed by an actor; *system action* if the action is performed by the system, or *use case action*, if the action consists of performing other use case, *i.e.* an use case *inclusion* or *extension* [7]. Actor actions and use case actions can be created only if some actor or another use case have been previously created. In general, objects can be created by means of context menus on potential parents or by means of the creation toolbar.

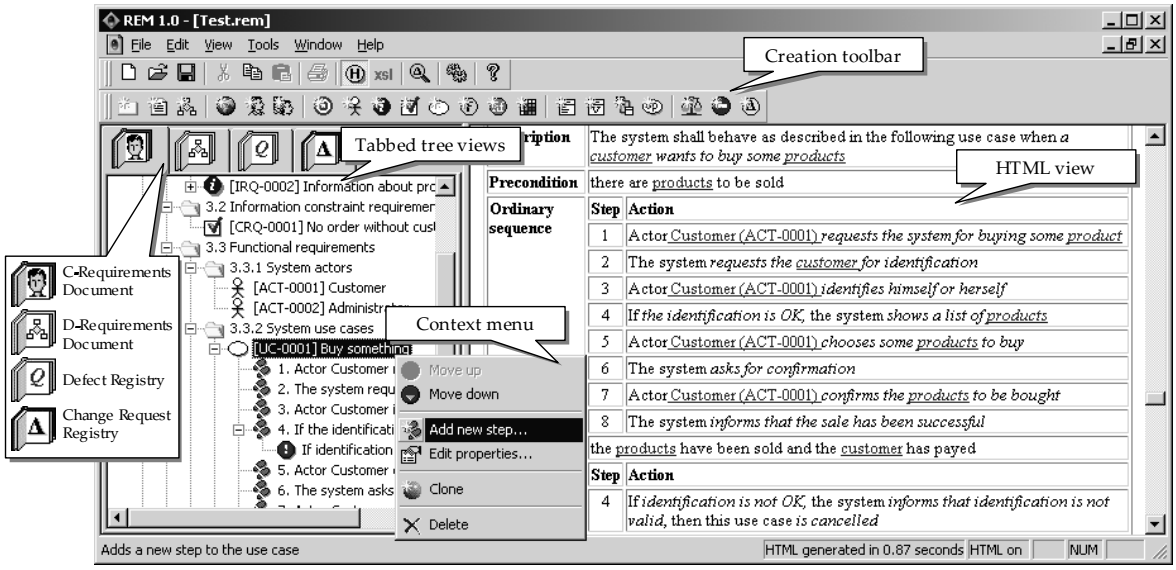

Figure 3: REM User Interface

4 XML Model of Requirements in REM

REM is based on an underlying object–oriented model of requirements described in [10] (a partial view of this model is shown in figures 4 and 4). The main object class of the model is the *Requirements Document*, that is composed of a sequence of REM objects (see figure 4 for a classification of REM objects).

We have translated our UML model of requirements into a relational schema and into a DTD. As an example, the UseCase class in figure 4 has been translated into the following DTD element definition:

```
<!ELEMENT rem:useCase (
  rem:name, rem:version,
  rem:authors?, rem:sources?, rem:comments?,
  rem:importance, rem:urgency, rem:status, rem:stability,
  rem:isAbstract?, rem:triggeringEvent,
  rem:precondition, rem:postcondition,
  rem:frequency, rem:step* )>
<!ATTLIST rem:useCase oid ID #REQUIRED>
```

Many of the elements in the previous DTD fragment (comments, triggeringEevent, pre and postcondition), contains only text, *i.e.* natural language. In REM, text can be composed of any combination of free text, references to other objects and TBD (*To Be Determined*) marks, defined as follows:

```
<!ELEMENT rem:text (#PCDATA|rem:ref|rem:tbd)*>
<!ELEMENT rem:ref (#PCDATA)>
   <!ATTLIST rem:ref oid IDREF #REQUIRED>
<!ELEMENT rem:tbd EMPTY>
```

where the rem:ref element must have a required attribute called oid that it is declared as an IDREF, *i.e.* a reference to other element with a matching identification attribute value. The concept of an IDREF attribute is very similar to the *foreign key* concept in relational databases.

The rem:tbd element is declared as an EMPTY element, *i.e.* it cannot have neither subordinate elements nor data. It is simply a mark. The DTD elements corresponding to use case steps and actions of figure 4 have been described as follows:

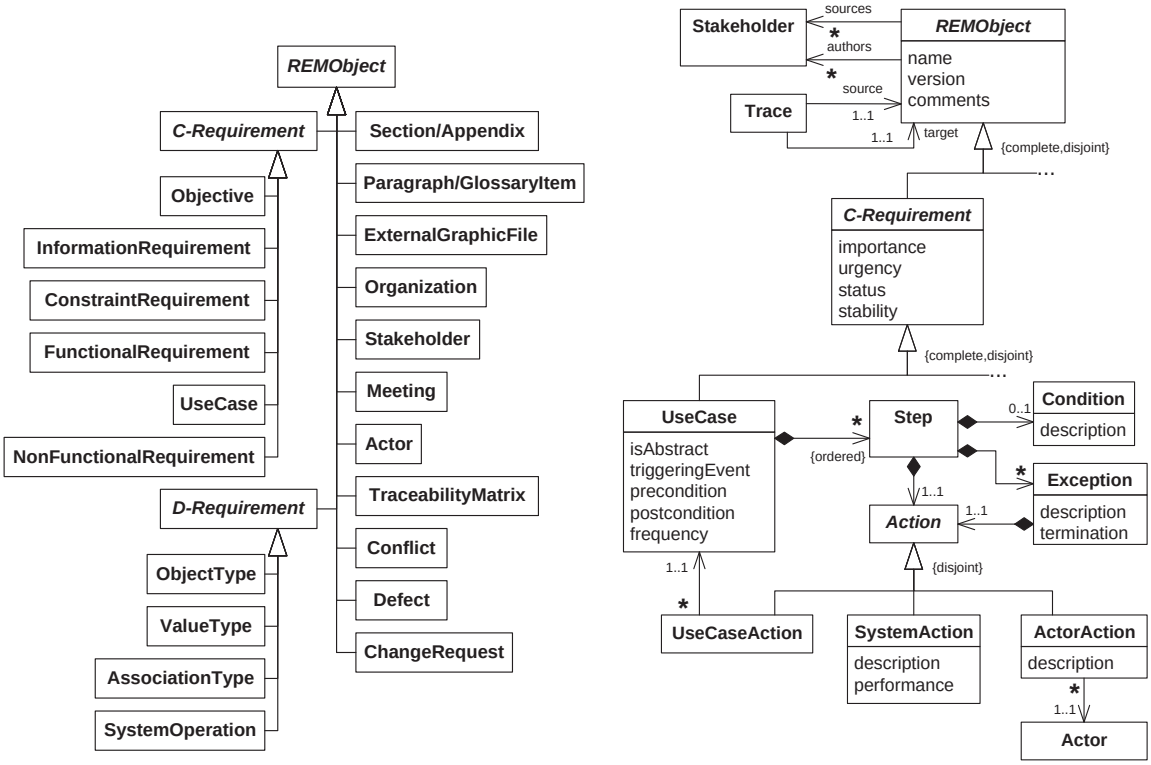

Figure 4: Partial UML models of REM

```
<!ELEMENT rem:step (
  rem:number, rem:condition?,
  ( rem:systemAction | rem:actorAction | rem:useCaseAction ), (rem:stepException*),
  rem:comments )>
<!ATTLIST rem:step oid ID #REQUIRED>

<!ELEMENT rem:systemAction ( rem:description, rem:performance? )>

<!ELEMENT rem:actorAction (rem:description)>
    <!ATTLIST rem:actorAction actor IDREF #REQUIRED>

<!ELEMENT rem:useCaseAction EMPTY>
    <!ATTLIST rem:useCaseAction useCase IDREF #REQUIRED>

<!ELEMENT rem:stepException (
  rem:condition,
  ( rem:systemAction | rem:actorAction | rem:useCaseAction ),  rem:termination,
  rem:comments )>
<!ATTLIST rem:stepException oid ID #REQUIRED>
```

Elements not defined in the previous DTD code (condition, description, termination, etc.) are defined as containing only text. For example:

```
<!ELEMENT rem:condition (#PCDATA|rem:ref|rem:tbd)*>
```

5 Using XSLT as a Requirements Verification Language

In the following sections we describe how some of the quality factors described in [12] can be automatically verified using XSLT when requirements are electronically stored in XML format according to the REM DTD.

5.1 Unambiguity

A requirement is unambiguous if and only if has only one possible interpretation [1]. Obviously, this is a semantic property and cannot be automatically verified, but we can provide some heuristics about potential ambiguities in requirements in order to focus verification effort on *potentially ambiguous requirements*.

A simple yet powerful heuristic for detecting ambiguity is looking for *weak phrases indicators* (WPIs) in requirements descriptions, *i.e.* "clauses that are apt to cause uncertainty and leave room for multiple interpretations", as described in [21]. This can be achieved with XSLT by means of the XPath [2] contains function, as in the following example:

```
<xsl:for-each select="//rem:useCase[contains(., 'easy') or contains(., 'not difficult') or . . . ]">
  Use case <xsl:value-of select="rem:name"/> contains some WPIs
</xsl:for-each>
```

where the xsl:for-each element selects all use cases with some WPI in their text. The dot notation used as the first argument of contains represents the text of the current node (an use case in the example) and of all its descendants. Although quite simple, previous XLST code is not very flexible, since the hard coding of WPIs makes the code very sensible to changes. This problem can be avoided by getting WPIs from an external XML file like this (wpi.xml):

```
<?xml version="1.0"?> <wpis>
   <wpi>easy</wpi>
   <wpi>etc</wpi>
   <wpi>normal</wpi>
   <wpi>not difficult</wpi>
   <wpi>timely</wpi>
   . . .
</wpis>
```

and using the following XSLT code:

```
<xsl:for-each select="//rem:useCase">
   <xsl:variable name="uc" select="current()"/>
   <xsl:for-each select="document('wpi.xml')//wpi">
      <xsl:if test="contains($uc,.)">
         Use case <xsl:value-of select="$uc/rem:name"/> contains WPI <xsl:value-of select="."/>
      </xsl:if>
   </xsl:for-each>
</xsl:for-each>
```

where every use case is checked against every WPI in wpi.xml using the XPath function document. Notice that the uc variable is needed since XSLT does not provide any way of accessing more than one current node at different levels in nested xsl:for-each structures.

Another complementary approach for ambiguity heuristics is measuring how much the customer's vocabulary is used in requirements descriptions. We agree with Leite [17] in the importance of understanding the language of the problem and in the importance of building a glossary (called *Language Extended Lexicon*, LEL, in [17]) at the beginning of the elicitation activity. Following Leite, two principles should be followed: the *principle of circularity*, (the glossary must be as self–contained as possible) and the *principle of minimal vocabulary*

(requirements descriptions must use as many glossary items as possible). Leite's principles cannot guarantee unambiguity, but they help to build more unambiguous, understandable, consistent, concise, and cross–referenced requirements [12], and can be used to measure the quality of the glossary and to detect potentially ambiguous requirements.

In the REM XML model, glossary items elements are mainly composed by rem:text elements[2] (see section 4), so they can have references to other REM objects as children. In this context, XSLT can be used to measure glossary circularity (GLC) and minimality of vocabulary (MOV). GLC can be measured as the ratio between the number of references to glossary items from other glossary items and the number of glossary items, as shown in the following XSLT code:

```
<xsl:variable name = "GLC"
   select = "count(//rem:glossaryItem//rem:ref[@oid = //rem:glossaryItem/@oid]) div count(//rem:glossaryItem)"
/>
```

where the XPath expression //rem:glossaryItem//rem:ref [@oid=//rem:glossaryItem/@oid] means "any rem:ref element descendant of some rem:glossaryItem, with an oid attribute corresponding to some oid attribute of some rem:glossaryItem element".

GLC can be used as an indicator of the glossary quality. GLC values under 1 indicate a low quality glossary, since that implies that there are glossary items not referencing other glossary items. GLC can also be computed for single glossary items as the number of references to other glossary items. It seems clear that glossary items not referencing other glossary items, or referencing just a few ones (less than 2, for example), should be verified for potential problems. The following XSLT code can be used for that purpose:

```
<xsl:for-each select="//rem:glossaryItem[count(.//rem:ref[@oid=//rem:glossaryItem/@oid]) < 2]">
   Please, check definition of glossary item <xsl:value-of select="rem:name"/>
</xsl:for-each>
```

MOV can be computed, in a similar way to GLC, as the ratio between the number of references to glossary items in requirements and the number of requirements. MOV can be used to pinpoint those requirements that do not have any reference, or just a few (less than 4, for example), to any glossary item in their text. Since those requirements are not using the vocabulary of the customer, they should be checked for potential problems of ambiguity or understandability [12]. The same schema used for detecting suspicious glossary items can also be used for detecting potentially ambiguous requirements:

```
<xsl:for-each select="//rem:useCase[count(.//rem:ref[@oid = //rem:glossaryItem/@oid]) < 4]">
   Please, check use case <xsl:value-of select="rem:name"/>, which use too few glossary items in its text
</xsl:for-each>
```

5.2 Completeness

A requirements document is complete if it includes [12]:

1. Everything that the software is supposed to do, *i.e. all* the requirements
2. Responses of the software to all classes of input data in all realizable situations
3. Page numbers, figure and table names and references, a glossary, units of measure and referenced material
4. No sections marked as TBD

The two first completeness conditions have to do with semantic properties of requirements and are therefore out of the scope of our approach. On the other hand, the third completeness condition is partially satisfied by

[2] Glossary items also have rem:name, rem:authors, etc. See the REM DTD for details.

means of the *correct–by–construction* paradigm of REM: figure and table names are automatically generated, references are automatically inserted and updated, and the user can easily create a glossary. If we want to be sure about the existence of a section named Glossary, we can apply the following XSLT code:

```
<xsl:choose>
  <xsl:when test="//rem:section[rem:name='Glossary']"/>
    There is a glossary
  </xsl:when>
  <xsl:otherwise>
    There is no glossary
  </xsl:otherwise>
</xsl:choose>
```

where the structure formed by xsl:choose, xsl:when and xsl:otherwise is basically an if–else–endif statement with multiple else branches. Notice that if we want to check the existence of an element we cannot use an XSLT template. If there is no such an element, the template will never match and we will have no output.

Similar XSLT code can be used to verify if requirements documents are *organized* [12], *i.e.* if they have mandatory sections in the mandatory order with mandatory content.

The fourth condition of completeness, the absence of TBD marks, can be easily verified using XSLT. If we want to know how many TBD marks are in a requirements document we can apply the following XLST code:

```
There are <xsl:value-of select="count(//rem:tbd)"/> TBD marks
```

that would generate in the output the number of occurrences of elements of type rem:tbd anywhere in the XML data. If we want to be more precise and we want to know, for example, what use cases have TBD marks inside their text and how many TBD marks they have, we can write the following XSLT code:

```
<xsl:template match="rem:useCase[.//rem:tbd]"/>
  Use case <xsl:value-of select="rem:name"/>
  has <xsl:value-of select="count(.//rem:tbd)"/> TBD marks
</xsl:template>
```

in which the select expression "rem:useCase[.//rem:tbd]" means "any use case with at least one descendant element of type rem:tbd".

5.3 Traceability

In [12], a requirements document is said to be *traceable* if and only if it is written in a manner that facilitates the referencing of each individual requirement. Since REM assigns automatically an unique identifier to every requirement (the required identifier attribute oid, see the DTD for use cases in section 4 as an example), this quality factor does not have to be verified explicitly.

What it must be checked is if the origin of every requirement is clear, *i.e.* if requirements are *traced* in the sense described in [12]. In our UML model of requirements, any REM object can be traced to and from other REM objects, including their human sources and authors (see figure 4). Checking if a requirement has sources and authors and if it is traced to or from other requirements is easy with XSLT. For example, the following XSLT template will match all use cases with no human sources:

```
<xsl:template match="rem:useCase[not(rem:sources)]">
  Use case <xsl:value-of select="rem:name"/> has no defined sources
</xsl:template>
```

And this XSLT template will match all non functional requirements not traced to other REM objects:

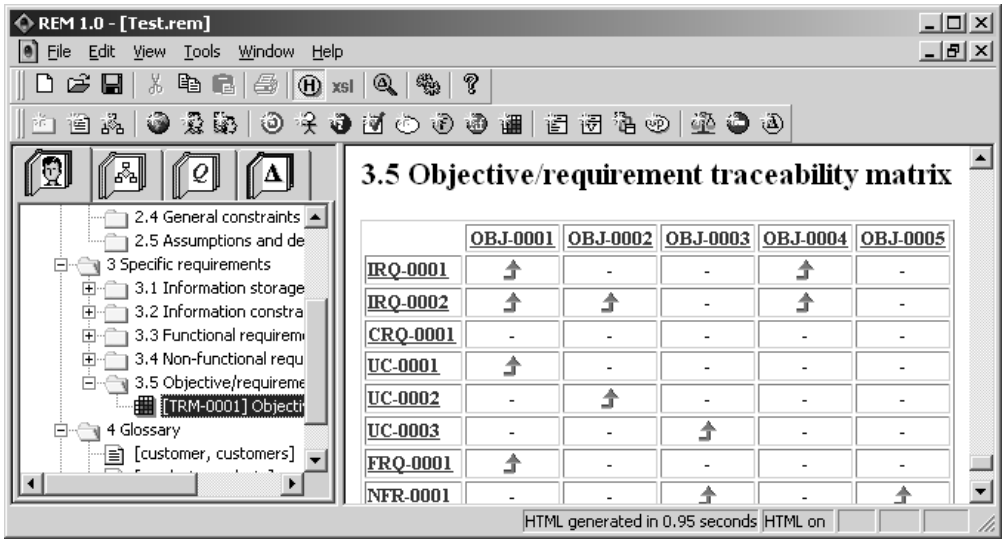

Figure 5: REM Traceability matrix example

```
<xsl:template match="rem:nonFunctionalRequirement">
  <xsl:if test="not(//rem:trace[@source=current()/@oid])">
    Non functional requirement <xsl:value-of select="rem:name"/> is not traced to any object
  </xsl:if>
</xsl:template>
```

In REM, traces are defined as elements with two required attributes of type IDREF, namely source and target. REM users can also use traceability matrices for visual checking of non–traced requirements (see figure 5).

5.4 Other verifiable quality factors

Applying the same ideas, other quality factors defined in [12] can be verified using XSLT, for example:

- What requirements are not annotated with relative importance, relative stability or version.
- What requirements have potentially ambiguous words in their description, like *easy to*, *user–friendly*, etc. by means of XSLT string functions like contains [3].
- If use cases are not well structured, *i.e.* if there are too few or too many *includes* or *extends* relationships.
- What use cases have too few [16] or too many steps, or too much exceptions, *i.e.* too many alternative courses .
- What actors do not participate in any use case [16].

6 Related work

Some of the results of the ESPRIT project CREWS, especially style and content guidelines for textual specification of use cases [20] have influenced our work. Their approach, implemented in the *CREWS-SAVRE* and *L'Ecritoire* tools, combines natural language processing (NLP) techniques and linguistic patterns, guiding the construction of use cases and providing a, much more powerful than ours, ambiguity detection mechanisms.

We use linguistic patterns extensively [11] and our UML model of requirements is partially based on CREWS results, but we have not adopted an NLP–based approach for verification because one of our goals

was to let REM users write their own verification programs (*i.e.* XSLT stylesheets). Using an NLP–based verification approach would have made this goal very difficult to satisfy because of their complexity. We are currently conducting experiments with our students in order to know if our simpler approach yields comparable results to the ones described in [5].

The Automated Requirement Measurement (ARM) tool [21], is simple yet powerful, not NLP–based requirements verification tool that scans requirements documents for specific words and phrases that are considered *indicators* of the quality of requirements. As shown in section 5.1, REM can perform the same type of analysis using XSLT and apply a wider set of verification–oriented heuristics.

Schematron [14] is an XML–based language for specifying assertions on XPath expressions in XML documents. A Schematron document must be transformed into a XSLT stylesheet using the Schematron XSLT stylesheet and then applying the resulting stylesheet to the XML document being analyzed. A comparison of our approach and Schematron was already presented in section 5.2. Both approaches relies on XPath expressions, but ours needs only one XSLT transformation and provides greater flexibility for presenting results to users and for computing requirements–oriented metrics.

The *xlinkit* language [18] is an XML–based language for specifying consistency rules between XML documents using a restricted set of first order logic and XPath expressions. Most XSLT verification code presented in this paper can be expressed in *xlinkit*. For example the XSLT code in section 5.3 for detecting use cases not traced to other REM objects can be expressed in *xlinkit* in the following way:

```
<globalset id="$useCases" xpath="//rem:useCase"/>
<globalset id="$traces" xpath="//rem:trace"/>

<consistencyrule id="beTraced">
  <description>
   Every use case must be traced to some object, i.e. must be the source of some trace
  </description>
  <forall var="uc" in="$useCases">
    <exists var="t" in="$traces">
     <equal op1="$uc/@oid"
          op2="$t/@source" />
    </exists>
  </forall>
</consistencyrule>
```

that identifies all use cases no traced to any object as *inconsistent links*. Translating other XSLT fragments into *xlinkit* requires the use of the XPath function true (*xlinkit* only allows equality and inequality expressions on XPath expressions as predicates), and the expansion of variable expressions (*xlinkit* does not allow the declaration of local variables inside forall or exists elements). For example, the *xlinkit* code corresponding to the *Checking the number of steps* heuristic is the following:

```
<globalset id="$useCases" xpath="//rem:useCase"/>

<consistencyrule id="checkingNOS">
  <description>
   Every use case should have 2<=NOS<=20
  </description>
  <forall var="uc" in="$useCases">
    <equal op1="(count($uc/rem:step) &gt;= 2) and (count($uc/rem:step) &lt;= 20)"
         op2="true()" />
  </forall>
</consistencyrule>
```

REM verification stylesheets must be XSLT in order to be processed by the REM user interface and they must be as efficient as possible for fast visual feedback to the user. *xlinkit* rules must be processed by a Java program so that makes it incompatible with REM. On the other hand, *xlinkit* presents results as xlinks, something that is not currently supported by most web browsers.

7 Conclusions and future work

In this paper we have presented an automated, light–weight, XSLT–based approach for the verification of natural language software requirements integrated in the REM requirements management tool. Our approach is based on a open technology like XML and offers all the flexibility of XSLT. In fact, if requirements are represented in XML not using the REM DTD, many of the XSLT–based verification–oriented heuristics presented in this paper should be easily adapted.

Our future work is focused in identifying accurate range values for those metrics–based verification heuristics (glossary–based and use case heuristics mainly). We are currently conducting an experiment with our students at the University of Seville with two main goals in mind. The first goal is determining range values by applying data mining techniques to our students' RE projects, thus identifying correlations between defects and metric values. The second goal is to know whether our approach improves requirements verification or not, *i.e.* if more defects in requirements are detected when requirements verification is supported by our XSLT–based heuristics.

References

[1] IEEE Recommended Practice for Software Requirements Specifications. IEEE/ANSI Standard 830–1998, 1998.

[2] XML Path Language (XPath) 1.0. W3C Recommendation, November 1999.

[3] XSL Transformations (XSLT) 1.0. W3C Recommendation, November 1999.

[4] Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000.

[5] C. Ben Achour, C. Rolland, N. A. M. Maiden, and C. Souveyet. Guiding Use Case Authoring: Results of an Empirical Study. In *ISRE'99 Proceedings*, 1999.

[6] B. W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, 1984.

[7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, 1999.

[8] J. W. Brackett. Software Requirements. Curriculum Module SEI–CM–19–1.2, Software Engineering Institute, Carnegie Mellon University, 1990.

[9] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison–Wesley, 1999.

[10] A. Durán. *A Methodological Framework for Requirements Engineering of Information Systems (in Spanish)*. PhD thesis, University of Seville, 2000.

[11] A. Durán, B. Bernárdez, A. Ruiz, and M. Toro. A Requirements Elicitation Approach Based in Templates and Patterns. In *WER'99 Proceedings*, Buenos Aires, 1999.

[12] A. Davis *et al.* Identifying and Measuring Quality in a Software Requirements Specification. In *Proceedings of the 1st International Software Metrics Symposium*, pages 141–152, 1993.

[13] F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi, and S. Ruggieri. Achieving Quality in Natural Language Requirements. In *Proceedings of the 11 th International Software Quality Week*, 1998.

[14] Rick Jelliffe. The Schematron Assertion Language 1.5. Technical report, Academia Sinica Computing Centre, 2001.

[15] G. Kontoya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, 1997.

[16] J. C. S. P. Leite, H. Hadad, J. Doorn, and G. Kaplan. A Scenario Construction Process. *Requirements Engineering Jornal*, 5(1), 2000.

[17] J. C. S. P. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad, and A. Oliveros. Enhancing a Requirements Baseline with Scenarios. In *Proceedings of the 3rd IEEE ISRE (RE'97)*, 1997.

[18] C. Nentwich, W. Emmerich, and A. Finkesltein. Static consistency checking for distributed specifications. In *Proceedings of Automated Software Engineering*, 2001.

[19] K. Pohl. Requirements Engineering: An Overview. *Encyclopedia of Computer Science and Technology*, 36, 1997.

[20] C. Rolland and C. Ben Achour. Guiding the Construction of Textual Use Case Specifications. *Data & Knowledge Engineering Journal*, 25(1–2), 1998.

[21] L. Rosenberg, T. F. Hammer, and L. L. Huffman. Requirements, Testing and Metrics. In *15th Annual Pacific Northwest Software Quality Conference*, Utah, 1998.